Shakti C-Class Multicore Verification

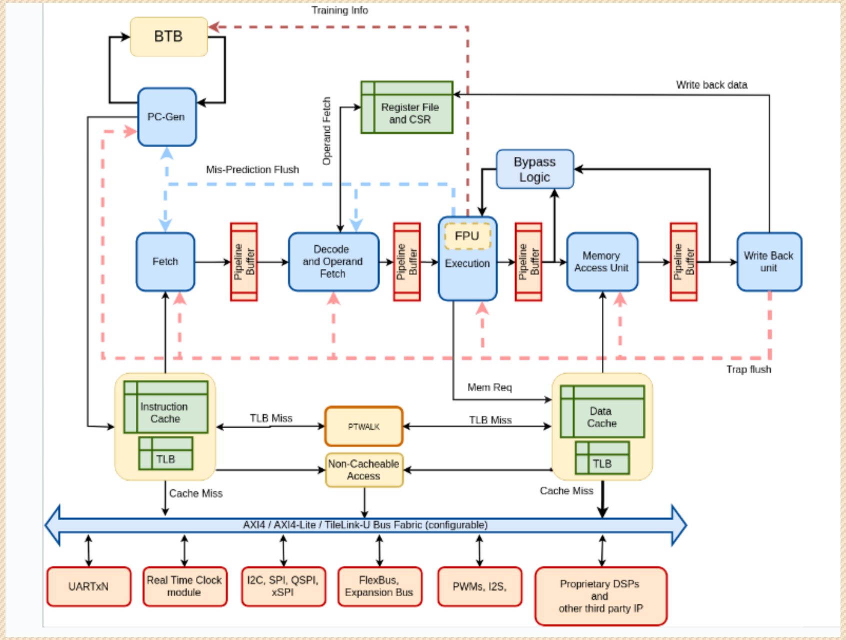
By Megna P

EE21B085

Mentored by Lavanya J.
Guide : Prof. Kamakoti V.

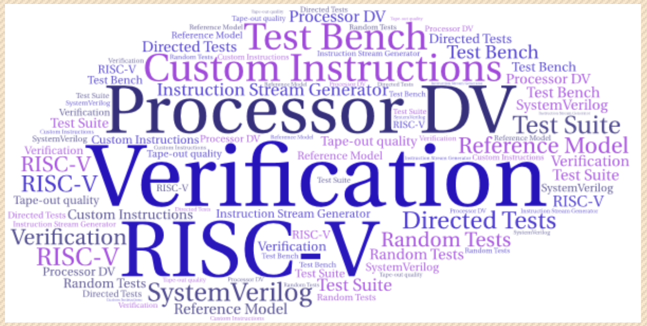
Shakti C-Class Multicore

- Shakti C-Class is a controller class processor with a 5-stage in-order design
- The processor supports RV64GCSUN ISA standard
- Multicore C-Class is built with 4 cores
- This configuration uses the OpenSMART Network-on-Chip (NoC) with Criticality
- MSI protocol is used for cache coherency



Why is Verification needed?

- Verification helps identify and fix bugs that leads to unexpected behavior and results
- Verification also provides reliability of the final product
- For multicore verification we need to verify the following:
  - Execution of various instruction streams
  - Ensuring functionality for multi-core setup using multi-hart tests
  - Coverage of various instructions for each test
- For these reasons, RISCV-DV tests are used to verify the Shakti C-Class Multicore



RISCV-DV tests

- RISCV-DV is an open-source, Python-based instruction generator designed for verification of RISC-V processors
- It supports multiple harts and generates tests to produce assembly programs
- RISCV-DV tests also offer co-simulation support with Spike, an Instruction Set Simulator (ISS)
- For verification, the simulations using Spike (which simulates an ideal RISC-V core) are compared with simulations using Shakti multicore

Features of RISCV-DV tests

- Generates various types of tests –
  - There are several types of tests in *baselist.yaml and these* generate specific instructions
  - Used to verify the processor’s functionality like arithmetic, logical and control flow operations
- Supports different targets –
  - RISCV-DV has support for various ISA like RV32I, RV64GC, etc
  - It has support for multi-harts too

Features of RISCV-DV tests

- Sub programs –
  - Subprograms are generated for each thread in multi-hart tests (h0_sub0, h0_sub1, etc.)
  - RISCV-DV can generate and call sub programs to verify control flow execution
- Coverage reports –
  - RISCV—DV can also monitor the coverage for a test and provides a metric for how thoroughly you've tested the design
  - For example, arithmetic tests generates assembly instructions for basic arithmetic operations, but coverage is zero for floating point operations

Generating Multi-hart tests using PyFlow simulator

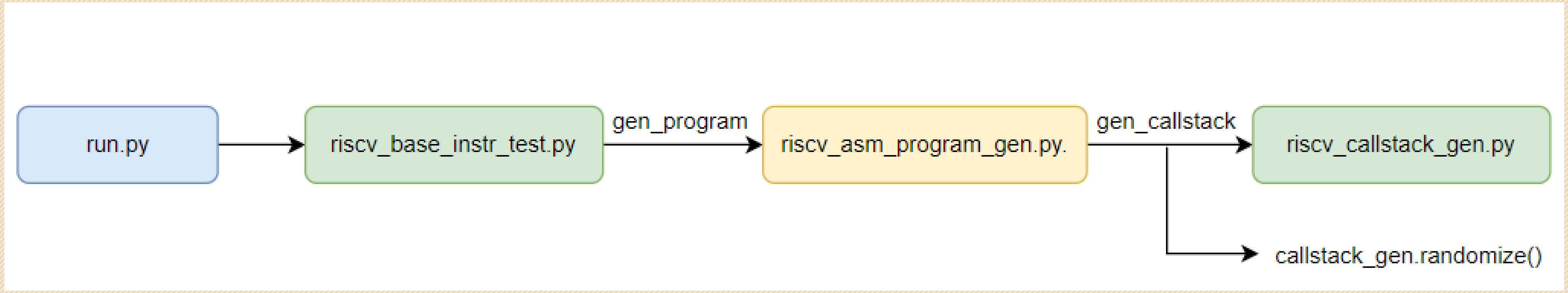
- *PyFlow* simulator invokes the Python instruction generation
- Instructions streams are assembly instructions which are generated specific to each test
- For multi-hart tests, assembly instructions are separately generated for each thread. (h0_main, h1_main, etc.)
- Multi-hart tests can be generated by modifying NUM_HARTS value as 4
- I have used two tests :
  - RISCV_ARITHMETIC_BASIC_TEST
  - RISCV_JUMP_STRESS_TEST

Sub programs

- For verifying multi-hart cores we need tests with multiple threads getting invoked
- With subprograms instructions are generated for each thread (h0_sub0, h0_sub1, etc.)
- Unfortunately, RISCV-DV tool had bugs and subprograms were not functional
- Issue with the tests was when the argument *num_of_sub_programs* was non-zero
- Test generation failed and assembly programs were not generated
- The fix is provided in the later slides

Sub programs

- The flow of function calls look like this :



- The class *riscv_asm_program_gen()* has the function *gen_program* which generates the main program
- It calls another function *gen_callstack*.
- An object named *callstack_gen* points to the class *riscv_callstack_gen()*.

Sub programs

- There are two specific functions defined inside the class *riscv_callstack_gen* which caused the issue:
  1. *problem_definition()* - This function is used to create a range of sub program IDs.
  2. *randomize()* - This function is used to randomize the unique sub program IDs.
- *gen_callstack.randomize() was* failing because *problem_definition()* isn't called before using *randomize()*
- Thus, it creates error trying to randomize an empty list
- After calling the function *problem_definition()* before *randomize()*, the error got fixed and the test generated sub programs for every hart

RISCV_ARITHMETIC_BASIC_TEST

- It is specifically designed to generate assembly instructions for basic arithmetic operations (add, sub, mul, div)
- The following command is used to generate arithmetic basic tests using *PyFlow:*

```
python3 run.py --test=riscv_arithmetic_basic_test –simulator=pyflow --steps gen
```

- Based on *num_of_tests*, it will generate that many number of assembly tests with given instruction count named as *riscv_arithmetic_basic_test_0.S* and *riscv_arithmetic_basic_test_1.S* and so on

RISCV_JUMP_STRESS_TEST

- This test generates a stressful amount of assembly instructions for various jump types supported by the target like unconditional (jal, jalr) and conditional (beq, bne, blt) jumps
- The following command is used to generate jump stress tests using *Pyflow:*

```
python3 run.py --test=riscv_jump_stress _test --simulator=pyflow --steps gen
```

- Based on *num_of_tests*, it will generate that many number of assembly tests with given instruction count named as *riscv_jump_stress_test_0.S* and *riscv_jump_stress_test_1.S* and so on

Spike simulation

- Spike simulation results are used to compare with Shakti's results for verification
- The generated assembly file is converted into a binary using this command:

```
riscv64-unknown-elf-gcc -nostartfiles -T link.ld riscv_arithmetic_basic_test_0.S
```

- This command will create *a.out*. This file can be used to run on spike using the following command –

```
spike -l --log=spike.log -p4 a.out  # -p4 is a flag to simulate 4 cores
```

Spike simulation

- The previous command will create the log file named as *spike.log*.
- After running on spike, we can generate a dump file which can be used for comparison with the shakti’s dump file. The command use is –

```
spike -l --log=spike.dump -p4 a.out
```

Simulation on Shakti Multicore

- Assembly tests cannot be directly run on Shakti Multicore
- These tests are converted into hex files using the *elf2hex* command:

```
elf2hex 8 33554432 a.out 2147483648 > code.mem
```

- *code.mem* should be present in the same folder as the Shakti executable. Then the following command will generate dump of shakti’s execution:

```
./out +rtldump
```

- Assembly instructions are executed on shakti multicore setup and *rtl.0 dump*, *rtl.1 dump*, *rtl.2 dump*, *rtl.3.dump* files are generated for the respective cores.

Spike dump vs Shakti multicore dump

- Shakti multicore's dump is compared against the Spike's dump files
- How is the comparison of dump files done?
  - *spike.dump* generated will have memory traces of all 4 cores.
  - A python script, *gendump.py,* was made *to* separately obtain memory dump for each core
  - Now the memory dump file for each core is compared using another python script *diff.py* to compare and store the difference in another file.

Comparison results

- CSR registers have mismatches as the machine instructions are dependent on the CPU architecture
- It will be conveyed to the design team for debugging to confirm if it is a RTL bug or not
- The following type of instructions were tested and functioned as expected:
    - Arithmetic instructions
    - Conditional branching
    - Unconditional jumps

Coverage reports

- Coverage reports give the metrics of coverage for a specific functionality of a processor
- Coverage for *riscv_arithmetic_basic_test*:
  - The spike.log file is used to create the coverage report using the following command –

    ```
    python3 cov.py –dir arithmetic_basic_test/asm_test/ --simulator pyflow –enable
    _visualization
    ```
  - In the *riscv_arithmetic_basic_test* all basic arithmetic instructions are used except for floating point instructions and jump instructions
  - Coverage scores for floating and jump instrucions are 0

Coverage reports

- Coverage for *riscv_jump_stress_test*:
  - The spike.log file is used to create the coverage report using the following command –

```
python3 cov.py –dir jump_test/asm_test/ --simulator pyflow –enable
_visualization
```

  - In *riscv_jump_stress_test* all jump instructions are used. The jump instruction scores are non-zero

Thank you